Java Abstract Methods - Complete Guide

What are Abstract Methods?

An **abstract method** is a method that is declared without implementation (no method body). It only has a method signature followed by a semicolon.

Key Characteristics:

- **No implementation** - just the method signature
- **Must be overridden** by subclasses
- **Cannot be instantiated** - class containing abstract methods becomes abstract
- **Forces implementation** - ensures subclasses provide their own version

Basic Syntax:

```java
// Abstract method syntax
access_modifier abstract return_type methodName(parameters);

// Examples:
public abstract void draw();
public abstract double calculateArea();
public abstract String getName();
protected abstract void process();
```

Why Use Abstract Methods?

1. **Enforce Implementation** - Force subclasses to implement specific methods
2. **Common Interface** - Ensure all subclasses have the same method signatures
3. **Polymorphism** - Allow different implementations while maintaining same interface
4. **Design Template** - Define what methods must exist without specifying how

Where Abstract Methods Can Exist:

1. Abstract Classes

```java
```

```java
abstract class Shape {
    protected String color;

    // Constructor (abstract classes can have constructors)
    public Shape(String color) {
        this.color = color;
    }

    // Concrete method (regular method with implementation)
    public void setColor(String color) {
        this.color = color;
    }

    // Abstract methods (no implementation)
    public abstract double calculateArea();
    public abstract double calculatePerimeter();
    public abstract void draw();

    // Another concrete method
    public void displayInfo() {
        System.out.println("Color: " + color);
        System.out.println("Area: " + calculateArea());       // Calls abstract method
        System.out.println("Perimeter: " + calculatePerimeter());
    }
}
```

2. Interfaces

java

```java
interface Drawable {
    // All methods in interfaces are implicitly abstract (unless default/static)
    void draw();              // implicitly "public abstract void draw();"
    void resize(int factor);  // implicitly "public abstract void resize(int factor);"
    void rotate(double angle);
}
```

Complete Example: Abstract Class with Abstract Methods

java

```java
// Abstract class
abstract class Animal {
    protected String name;
    protected int age;

    // Constructor
    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Concrete methods (have implementation)
    public void sleep() {
        System.out.println(name + " is sleeping");
    }

    public void eat() {
        System.out.println(name + " is eating");
    }

    // Abstract methods (no implementation - must be overridden)
    public abstract void makeSound();
    public abstract void move();
    public abstract String getSpecies();

    // Concrete method that uses abstract methods
    public void performActions() {
        System.out.println("=== " + name + " (" + getSpecies() + ") ===");
        eat();
        makeSound();
        move();
        sleep();
    }
}

// Concrete subclass - must implement all abstract methods
class Dog extends Animal {
    private String breed;

    public Dog(String name, int age, String breed) {
        super(name, age);    // Call parent constructor
        this.breed = breed;
    }

    // Must implement all abstract methods from parent
    @Override
```

```
    public void makeSound() {
        System.out.println(name + " barks: Woof! Woof!");
    }

    @Override
    public void move() {
        System.out.println(name + " runs on four legs");
    }

    @Override
    public String getSpecies() {
        return "Canine - " + breed;
    }

    // Dog-specific method
    public void wagTail() {
        System.out.println(name + " is wagging tail");
    }
}

class Bird extends Animal {
    private boolean canFly;

    public Bird(String name, int age, boolean canFly) {
        super(name, age);
        this.canFly = canFly;
    }

    // Must implement all abstract methods
    @Override
    public void makeSound() {
        System.out.println(name + " chirps: Tweet! Tweet!");
    }

    @Override
    public void move() {
        if (canFly) {
            System.out.println(name + " flies in the sky");
        } else {
            System.out.println(name + " walks on the ground");
        }
    }

    @Override
    public String getSpecies() {
        return "Avian" + (canFly ? " (Flying)" : " (Flightless)");
    }
```

```
}

class Fish extends Animal {
    private String waterType;

    public Fish(String name, int age, String waterType) {
        super(name, age);
        this.waterType = waterType;
    }

    @Override
    public void makeSound() {
        System.out.println(name + " makes bubbling sounds");
    }

    @Override
    public void move() {
        System.out.println(name + " swims in " + waterType + " water");
    }

    @Override
    public String getSpecies() {
        return "Aquatic (" + waterType + "water)";
    }

    // Override inherited method with specific behavior
    @Override
    public void sleep() {
        System.out.println(name + " sleeps while swimming slowly");
    }
}
```

Interface with Abstract Methods Example:

java

```java
// Interface - all methods are implicitly abstract
interface Vehicle {
    // Abstract methods (implicitly public abstract)
    void start();
    void stop();
    void accelerate();
    double getFuelEfficiency();

    // Default method (Java 8+) - has implementation
    default void displayInfo() {
        System.out.println("Fuel Efficiency: " + getFuelEfficiency() + " km/l");
    }
}

interface ElectricVehicle extends Vehicle {
    // Additional abstract methods
    void chargeBattery();
    double getBatteryLevel();

    // Override default method
    @Override
    default void displayInfo() {
        System.out.println("Battery Level: " + getBatteryLevel() + "%");
        System.out.println("Fuel Efficiency: " + getFuelEfficiency() + " km/kWh");
    }
}

// Implementing interface - must implement all abstract methods
class Car implements Vehicle {
    private String brand;
    private boolean isRunning;

    public Car(String brand) {
        this.brand = brand;
        this.isRunning = false;
    }

    @Override
    public void start() {
        isRunning = true;
        System.out.println(brand + " car is starting with ignition");
    }

    @Override
    public void stop() {
        isRunning = false;
```

```
        System.out.println(brand + " car is stopping");
    }

    @Override
    public void accelerate() {
        if (isRunning) {
            System.out.println(brand + " car is accelerating");
        } else {
            System.out.println("Start the car first!");
        }
    }

    @Override
    public double getFuelEfficiency() {
        return 15.5; // km per liter
    }
}

class ElectricCar implements ElectricVehicle {
    private String brand;
    private double batteryLevel;
    private boolean isRunning;

    public ElectricCar(String brand) {
        this.brand = brand;
        this.batteryLevel = 100.0;
        this.isRunning = false;
    }

    // Implement Vehicle methods
    @Override
    public void start() {
        if (batteryLevel > 0) {
            isRunning = true;
            System.out.println(brand + " electric car is starting silently");
        } else {
            System.out.println("Battery is empty! Charge first.");
        }
    }

    @Override
    public void stop() {
        isRunning = false;
        System.out.println(brand + " electric car is stopping");
    }

    @Override
```

```java
    public void accelerate() {
        if (isRunning) {
            batteryLevel -= 1.0;
            System.out.println(brand + " electric car is accelerating smoothly");
        } else {
            System.out.println("Start the car first!");
        }
    }

    @Override
    public double getFuelEfficiency() {
        return 25.0; // km per kWh
    }

    // Implement ElectricVehicle methods
    @Override
    public void chargeBattery() {
        batteryLevel = 100.0;
        System.out.println(brand + " battery charged to 100%");
    }

    @Override
    public double getBatteryLevel() {
        return batteryLevel;
    }
}
```

Testing Abstract Methods:

java

```
public class AbstractMethodDemo {
    public static void main(String[] args) {
        // Cannot instantiate abstract class
        // Animal animal = new Animal("Generic", 5); // Compilation Error

        System.out.println("=== TESTING ABSTRACT CLASS ===");

        // Create concrete objects
        Dog dog = new Dog("Buddy", 3, "Golden Retriever");
        Bird bird = new Bird("Tweety", 1, true);
        Fish fish = new Fish("Nemo", 2, "salt");

        // Call methods (including abstract methods implemented by subclasses)
        dog.performActions();
        dog.wagTail();

        System.out.println();
        bird.performActions();

        System.out.println();
        fish.performActions();

        System.out.println("\n=== TESTING INTERFACES ===");

        // Create objects implementing interfaces
        Car car = new Car("Toyota");
        ElectricCar eCar = new ElectricCar("Tesla");

        // Test regular car
        car.start();
        car.accelerate();
        car.displayInfo();
        car.stop();

        System.out.println();

        // Test electric car
        eCar.start();
        eCar.accelerate();
        eCar.displayInfo();
        eCar.chargeBattery();
        eCar.stop();

        System.out.println("\n=== POLYMORPHISM WITH ABSTRACT METHODS ===");

        // Polymorphism - treating different objects the same way
```

```
        Animal[] animals = {dog, bird, fish};

        for (Animal animal : animals) {
            System.out.println("\n--- " + animal.getClass().getSimpleName() + " ---");
            animal.makeSound();  // Calls overridden abstract method
            animal.move();       // Calls overridden abstract method
        }

        // Polymorphism with interfaces
        Vehicle[] vehicles = {car, eCar};

        System.out.println("\n=== VEHICLE POLYMORPHISM ===");
        for (Vehicle vehicle : vehicles) {
            vehicle.start();
            vehicle.accelerate();
            vehicle.displayInfo();
        }
    }
}
```

Rules for Abstract Methods:

✅ What You CAN Do:

1. **Declare in abstract classes**
2. **Declare in interfaces**
3. **Use any access modifier** (except private in interfaces)
4. **Have parameters and return types**
5. **Throw exceptions**
6. **Be overridden in subclasses**

❌ What You CANNOT Do:

1. **Have a method body** - no implementation
2. **Be static** - static methods must have implementation
3. **Be final** - final methods cannot be overridden
4. **Be private** - must be overridable by subclasses
5. **Exist in concrete classes** - class becomes abstract

Abstract Method Rules in Detail:

```
java
```

```
abstract class Example {
    // ✅ Valid abstract methods
    public abstract void method1();
    protected abstract void method2();
    abstract void method3();                    // default access
    public abstract String method4(int x);
    protected abstract void method5() throws Exception;

    // ❌ Invalid abstract methods
    // private abstract void method6();         // Cannot be private
    // static abstract void method7();          // Cannot be static
    // final abstract void method8();           // Cannot be final
    // public abstract void method9() { }       // Cannot have body
}
```

Benefits of Abstract Methods:

1. **Enforced Implementation**: Guarantees subclasses implement required methods
2. **Consistent Interface**: All subclasses have the same method signatures
3. **Polymorphism**: Can treat different objects uniformly
4. **Design Template**: Provides structure for inheritance hierarchy
5. **Code Organization**: Separates interface from implementation

Common Use Cases:

1. **Template Method Pattern**: Define algorithm structure, let subclasses implement details
2. **Strategy Pattern**: Different implementations of same interface
3. **Plugin Architecture**: Common interface for different plugins
4. **Framework Development**: Base classes that must be extended

Abstract Methods vs Concrete Methods:

Feature	Abstract Method	Concrete Method
Implementation	No body	Has body
Override	Must override	Can override
Class Type	Makes class abstract	Can be in any class
Instantiation	Cannot instantiate	Can instantiate
Purpose	Force implementation	Provide implementation

Key Points to Remember:

1. **Abstract methods have no implementation** - just signature + semicolon
2. **Classes with abstract methods must be declared abstract**
3. **Subclasses must implement all inherited abstract methods**
4. **Abstract classes cannot be instantiated**
5. **Interface methods are implicitly abstract** (unless default/static)
6. **Abstract methods enable polymorphism and consistent interfaces**
7. **Use abstract methods to create templates and enforce contracts**

Abstract methods are a powerful feature that helps create well-structured, maintainable object-oriented designs!